Entrées/sorties, compléments

I. Lecture/écriture dans un fichier texte

I.1. `with` et `open`

a) Exercice d'introduction

Le fichier « dictionnaire.dic » contient une liste de mots de la langue française.
Le code suivant ouvre le fichier en lecture,

```
with open('dictionnaire.dic','r',encoding='utf-8') as f:
    lignes=f.readlines()
mots=[mot[:-1] for mot in lignes]
```

La première ligne permet d'ouvrir le fichier « dictionnaire.dic » en lecture (paramètre `'r'` pour *read*).
La seconde lit toutes les lignes du fichier et stocke le résultat de la lecture dans une liste, chaque élément de la liste correspondant à une ligne du fichier ouvert.
La troisième enlève le dernier caractère de chaque ligne : en effet, le dernier caractère de chaque ligne est en fait le caractère `'\n'` de passage à une nouvelle ligne, et ce caractère ne fait évidemment jamais partie des mots stockés dans le dictionnaire.
Créer une fonction `anagramme(m)` qui trouve tous les anagrammes d'un mot `m` donné.
Variation : modifier votre fonction `anagramme(m)` de sorte à ce que les caractères accentués soient considérés comme identiques.
Plus compliqué : obtenir la liste de toutes les listes d'anagrammes du dictionnaire.

b) Ouverture/fermeture d'un fichier : syntaxe

Syntaxe : open

Pour ouvrir un fichier ***texte***, on utilise la fonction `open` du noyau Python. La fonction renvoie un ***objet*** qui représente le fichier au sein du code Python : ***c'est cet objet qui permettra de lire, d'écrire ou d'ajouter*** du texte dans le fichier. La syntaxe générale de la fonction `open` est la suivante :
`objet=open(nom,mode)`
où `nom` désigne le nom du fichier (éventuellement précédé de son chemin d'accès si le fichier n'est pas dans le même répertoire que le code Python exécuté)
et où `mode` est un caractère désignant la façon dont le fichier est ouvert :

- si `mode` vaut `'r'` (pour ***read***), on pourra lire dans le fichier ;
- si `mode` vaut `'w'` (pour ***write***), le fichier est effacé. On pourra ensuite écrire dans le fichier.
- Si `mode` vaut `'a'` (pour ***append***), le fichier est ouvert. On pourra ajouter du texte à la fin du le fichier.
- Si `mode` vaut `'+'`, le fichier est mis-à-jour s'il existait, créé sinon. Souvent associé à `'w'` et `'a'`.

Important ! Blocage de l'accès à un fichier

Nous avons vu au chapitre 1 section I.3. que le système d'exploitation assure entre autre la gestion des fichiers. Python fait directement appel au système d'exploitation pour l'ouverture d'un fichier et la fonction `open` n'est en fait qu'une interface permettant au programmeur de ne pas avoir à se soucier de la façon dont les divers systèmes d'exploitation gèrent les accès aux fichiers. Par exemple, la façon dont un texte est encodé sous forme binaire dépend du système d'exploitation : `utf-8` pour Linux et MacOS, `latin1` pour les versions de Windows antérieures à la version 7 mais `utf-8` à nouveau pour Windows 8...
Tout ceci est géré par la fonction `open` et le programmeur n'a donc pas à s'en préoccuper, ce qui permet aux codes écrits d'être indépendants du système d'exploitation sur lequel ils sont exécutés : on dit que le code est ***indépendant de la plateforme***.
Cependant, certains problèmes doivent absolument être pris en considération lors de l'accès à un fichier. Essentiellement, il faut comprendre que :

- pour ouvrir un fichier, encore faut-il avoir les ***droits d'accès requis par le système d'exploitation*** ;
- lorsqu'un fichier est ouvert par un programme, ***il ne peut plus l'être par un autre pour éviter que deux programmes apportent des modifications différentes et simultanées à un même fichier*** ce qui provoquerait ou bien des plantages machines, ou bien des erreurs sur les modifications attendues dans le fichier.

Si un fichier est ouvert par Python, il ne peut plus l'être par un autre programme (y compris un second code Python qu'on tenterait d'exécuter simultanément avec le premier).
C'est la raison pour laquelle ***un fichier ouvert doit absolument être fermé après utilisation !***
Deux alternatives sont possibles :

Syntaxe : close

C'est ***la méthode*** (au sens informatique du terme) permettant de fermer un fichier, c'est-à-dire de prévenir le système d'exploitation que le fichier ne sera plus utilisé par le code Python. Si `objet` désigne un fichier ouvert, la fermeture du fichier s'accomplit à l'aide de la syntaxe :
`objet.close()`

Une alternative (préférable) est l'emploi de l'instruction `with` :

Syntaxe : with

L'instruction `with` permet de fermer ***automatiquement*** le fichier dès que les tâches de lecture/écriture/modification sont accomplies. Parce que la fermeture s'accomplit automatiquement, cette instruction évite beaucoup d'erreurs qui sont - qui plus est - souvent irrattrapables : un fichier ouvert qui n'aurait pas été refermé ne pourra plus être ni lu ni modifié jusqu'au redémarrage de la machine.
Pour ouvrir un fichier on utilisera donc de préférence la syntaxe :

```
with open(nom,mode) as objet:
    BlocDeCodeIndente # le fichier est ouvert
PoursuiteDuCodeNonIndente # le fichier a été automatiquement ferme
                          # objet est inaccessible
```

I.2. Lecture/écriture

Une fois le fichier ouvert, on dispose essentiellement de deux méthodes pour lire/écrire dans le fichiers :

Syntaxe : Méthode `readlines`

`fichier.readlines()` permet de lire l'intégralité du fichier et renvoie son contenu dans une chaîne de caractères.
Les lignes sont séparées par le caractère spécial `'\n '`. Exemple :

```
with open("monfichier.txt",'r') as f:
    contenu = f.readlines()
```

Syntaxe : Méthode `writelines`

`fichier.writelines(chaine)` permet d'écrire le contenu de `chaine` dans le fichier. Pour passer à la ligne, on utilise le caractère spécial `'\n '`. Exemple :

```
chaine = 'un fichier sur\ndeux lignes'
with open("monfichier.txt",'w+') as f:
    f.writelines(chaine)
```

I.3. Le format `csv` et l'écriture d'un fichier texte

Syntaxe : Format `csv`

Le format `csv` est un format générique pour l'écriture de listes ou de tableaux dans un fichier texte.
On utilise pour cela deux séparateurs :
`'\n '` est utilisé comme séparateur des lignes du tableau tandis que
`','` ou `';'` est utilisé comme séparateur pour les colonnes à l'intérieur d'une même ligne.
La première ligne du fichier peut - éventuellement - contenir des informations générales sur le fichier, comme par exemple le titre des colonnes présentes dans le fichier.

Calculer, pour chaque lettre de l'alphabet, sa fréquence dans le dictionnaire « dictionnaire.dic ».
Écrire un fichier texte, nommé `"frequences.csv"` dont la première ligne est constituée des lettres de l'alphabet, séparées par des virgules, et dont la seconde ligne donne les fréquences d'apparition de chacune de ces lettres (séparées par des virgules).

II. Représentations graphiques

Comme nous l'avons vu au chapitre 4, aux TDs 15 et 14 et dans le devoir à la maison, les modules `numpy` et `matplotlib.pyplot` peuvent être utilisés pour faire divers types de représentations graphiques.

II.1. `numpy`

Ce module apporte essentiellement deux grandes améliorations au noyau Python :

1) un grand nombre de fonctions et constantes mathématiques ;

2) des objets de type `numpy.ndarray` permettant de créer des tableaux de nombres. Ces tableaux ont un fonctionnement similaire quoique différent des listes avec quelques avantages et quelques inconvénients :
   - les tableaux `numpy` doivent posséder des éléments de type identique ;
   - ce sont des tableaux d'ordre n ou $n \times p$ ou $n \times p \times q$ (où $n, p, q \in \mathbb{N}^*$) etc... Notamment, il est impossible de créer des tableaux triangulaires comme par exemple un triangle de Pascal ;
   - une opération effectuée sur un tableau `numpy` se concrétise en ***cette même opération effectuée sur chacun des éléments du tableau***. Par exemple, que donne le code suivant ?

     ```
     import numpy as np
     M=np.array([[i+3*j for i in range(3)] for j in range(3)])
     print((M+1)**2)
     ```
   - de même, une fonction d'une variable numérique appelée avec pour paramètre d'entrée un tableau de valeurs numériques renverra ***le tableau des images par cette fonction des valeurs du tableau d'entrée*** ;
   - à priori, ces tableaux sont faits pour conserver toujours le même nombre d'éléments - même s'il est en fait possible de redimensionner un tableau `numpy` pour y rajouter des éléments.

Voici quelques précisions supplémentaires :

a) Tableaux `numpy`

`numpy` définit des tableaux dont la syntaxe ressemble à celle des listes. La principale différence réside dans le fait que pour accéder à l'élément $a_{i,j}$ d'un tableau à 2 dimensions on écrira `a[i,j]` pour un tableau `numpy` alors qu'on aurait écrit, pour une liste, ...
On peut comme pour les listes utiliser la syntaxe de ***slicing*** dans un tableau `numpy`.

Syntaxe : transformation d'une liste en tableau `numpy` et vice versa

`np.array(liste)`
Prend une liste en argument et renvoie un tableau `numpy` ayant les même dimensions et les mêmes valeurs.
`tableau.tolist()`
Renvoie la liste ayant les mêmes dimensions et les mêmes valeurs que `tableau`.

Méthode : Créer un tableau `numpy`

Pour créer un tableau `numpy`, on utilisera l'une des syntaxes suivantes :
- on crée une liste `L` à l'aide des outils habituels de manipulation de liste puis le tableau `numpy` correspondant à l'aide de `np.array` ;
- on crée directement un tableau `numpy` aux bonnes dimensions et rempli soit de 0 soit de 1 à l'aide de `np.zeros` ou de `np.ones`.

```
>>> import numpy as np
>>> L=[[3*i+j for j in range(3)] for i in range(3)]
>>> T=np.array(L)
>>> print(T)
[[0 1 2]
 [3 4 5]
 [6 7 8]]
>>> T0=np.zeros((2,2))
```

```
>>> T1=np.ones((2,2,2))
>>> print(T0)
[[0. 0.]
 [0. 0.]]
>>> print(T1)
[[[1. 1.]
  [1. 1.]]

 [[1. 1.]
  [1. 1.]]]
```

b) Divers objets `numpy`

Syntaxe

`numpy.linspace(start,stop,num=50)`
Renvoie un tableau `numpy` à une dimension comprenant `num` éléments dont les valeurs sont régulièrement espacées de `start` à `stop` (inclus).
`numpy.linalg`
Sous-module regroupant divers outils de calcul matriciel.
Diverses fonctions mathématiques
`numpy.abs`, `numpy.arccos`, `numpy.arcsin`, `numpy.arctan`, `numpy.arctan2`, `numpy.cos`, `numpy.cosh`, `numpy.exp`, `numpy.floor`, `numpy.log`, `numpy.min`, `numpy.mean`, `numpy.max`, `numpy.sin`, `numpy.sinh`, `numpy.sort`, `numpy.tan`, etc...
On peut aussi citer `numpy.pi` qui est une constante donnant une valeur approchée de π.

II.2. `matplotlib`

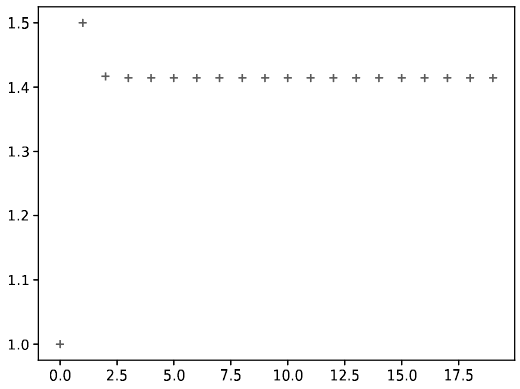
Le module `matplotlib` permet de faire divers types de représentations graphiques. Chargeons les modules `numpy` et `matplotlib.pyplot` :

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

`plt` désigne désormais le sous-module `matplotlib.pyplot` et `np` le module `numpy`.

II.3. Représentations graphiques de suites

```
>>> nb_pts=20
>>> abscisses=list(range(nb_pts))
>>> def Heron(n):
...     u=1
...     L=[u]
...     while len(L)<n:
...         u=(u+2/u)/2
...         L.append(u)
...     return L
...
>>> ordonnees=Heron(nb_pts)
>>> plt.plot(abscisses,ordonnees,'+')
[<matplotlib.lines.Line2D object at 0x000001D7E760FE88>]
```

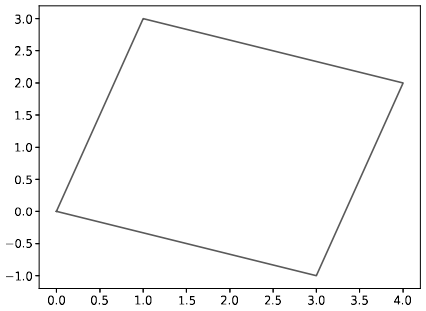


Dans le code précédent, on donne la liste des abscisses des points à afficher (ici $[\![0;19]\!]$), la liste des ordonnées de ces points (ici u_k pour $k \in [\![0;19]\!]$) puis on affiche ces points à l'aide de la syntaxe `plt.plot(abscisses,ordonnees,'+')` où le troisième paramètre `'+'` indique que l'on souhaite n'afficher que des points, et non pas les segments reliant ces points.

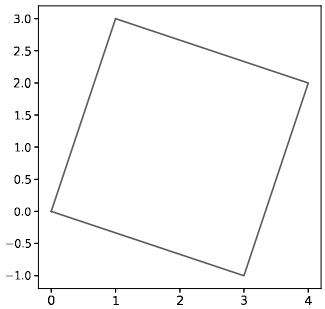
II.4. Ligne polygonale

C'est la même syntaxe que dans l'exemple précédent, mais sans le troisième paramètre : par défaut, la représentation graphique affichera des segments reliant chaque couple de deux points successifs.

```
>>> abscisses=[0,1,4,3,0]
>>> ordonnees=[0,3,2,-1,0]
>>> plt.plot(abscisses,ordonnees)
[<matplotlib.lines.Line2D object at 0x000001D7E8A1FF88>]
```



Les points donnés dans la représentation graphique précédente sont les sommets d'un carré.
La raison pour laquelle la représentation graphique n'est pas celle d'un carré est .....
.....
Pour orthonormer le repère, on utilise la syntaxe
`plt.axis('scaled')`



II.5. Représentations graphiques de fonctions

À nouveau, c'est exactement la même syntaxe. Simplement, pour que la représentation de la fonction paraisse lisse, on fera en sorte que les points de la ligne polygonale soient très rapprochés les uns des autres.
Pour avoir des abscisses uniformément répartis sur l'intervalle où l'on souhaite représenter la fonction, on utilisera de préférence `np.linspace(xm,xM,nbpoints)`.
Pour tracer plusieurs représentations graphiques dans un même repère, il suffit de faire plusieurs appels successifs à la méthode `plot`.
Ex. 7.1 Tracer, sur un même graphique, les représentations graphiques des fonctions $x \mapsto x$, ln et exp sur le segment $[-2;2]$.
Que devrait-on constater ?

Cor. 7.1

Ex. 7.2 On rappelle que pour une fonction f de I (intervalle symétrique par rapport à 0) dans $\mathbb{R}$, la ***partie paire*** de cette fonction est la fonction $u : \left\{ \begin{array}{ccc} I & \to & \mathbb{R} \\ x & \mapsto & \dfrac{f(x)+f(-x)}{2} \end{array} \right.$
et la ***partie impaire*** de f est la fonction $v : \left\{ \begin{array}{ccc} I & \to & \mathbb{R} \\ x & \mapsto & \dfrac{f(x)-f(-x)}{2} \end{array} \right.$

1) Écrire une fonction `RGPartiesPairesImpaires(f,a,b)` qui trace sur un même repère ***orthonormé*** les représentations graphiques de la ***fonction*** f sur l'intervalle $[a;b]$, de sa partie paire et de sa partie impaire.
2) Tester votre code en prenant $f = \text{Arccos}$, $a = -1$, $b = 1$.

Cor. 7.2

II.6. Représentations graphiques de tableaux 2D ou d'images

On utilise pour cela `plt.imshow(tableau)`.
Nous verrons cette possibilité en détail lors d'un TD.

II.7. Enregistrement/chargement d'images

Pour enregistrer une représentation graphique, on utilise
`plt.savefig('nomdufichier.jpg')`
Pour enregistrer une image contenue dans un tableau `numpy`, on utilise
`plt.imsave('nomdufichier.jpg',tableau)`
Enfin, pour charger une image, on utilise
`tableau=plt.imread('nomdufichier.jpg')`